

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

(2)

Public reporting burden
needed, and review
Headquarters Service
Management and Bu

AD-A245 444



including the time for reviewing instructions, searching existing data sources gathering and maintaining the data
or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington
Way, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of

1. AGENCY USE

3. REPORT TYPE AND DATES COVERED

Final: 06 March 1991

4. TITLE AND SUBTITLE

Rockwell International Corporation, DDC-Based Ada/CAPS Compiler, Version 6.0,
VAXstation 3100 Model 30 (under VMS 5.4) to CAPS/AAMP2 (bare
machine)(Target), 910306W1.11130

5. FUNDING NUMBERS

6. AUTHOR(S)

Wright-Patterson AFB, Dayton, OH
USA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Ada Validation Facility, Language Control Facility ASD/SCEL
Bldg. 676, Rm 135
Wright-Patterson AFB, Dayton, OH 45433

8. PERFORMING ORGANIZATION
REPORT NUMBER

AVF-VSR-0891

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Ada Joint Program Office
United States Department of Defense
Pentagon, Rm 3E114
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY
REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Rockwell International Corporation, DDC-Based Ada/CAPS Compiler, Wright-Patterson, AFB, Version 6.0, VAXstation
3100 Model 30 (under VMS 5.4) to CAPS/AAMP2 (bare machine)(Target), ACVC 1.11.

DTIC
ELECTE
JAN 14 1992
S B D

14. SUBJECT TERMS

Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val.
Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION
OF REPORT
UNCLASSIFIED

18. SECURITY CLASSIFICATION
UNCLASSIFIED

19. SECURITY CLASSIFICATION
OF ABSTRACT
UNCLASSIFIED

20. LIMITATION OF ABSTRACT

AVF Control Number: AVF-VSR-449-0891
21-August-1991
90-08-15-RWL

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 910306W1.11130
Rockwell International Corporation
DDC-Based Ada/CAPS Compiler, Version 6.0
VAXstation 3100, Model 30 => CAPS/AAMP2 (bare machine)

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

92 1 18 072

92-01158



Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 06 March 1991.

Compiler Name and Version: DDC-Based Ada/CAPS Compiler, Version 6.0

Host Computer System: VAXstation 3100 Model 30 (under VMS 5.4)


Target Computer System: CAPS/AAMP2 (bare machine)


Customer Agreement Number: 90-08-15-RWL


See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 910306W1.11130 is awarded to Rockwell International Corporation. This certificate expires on 1 June 1993.

This report has been reviewed and is approved.


Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503


for Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311


for Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

Attachment 2
Declaration of Conformance (AAMP2)

DECLARATION OF CONFORMANCE

Customer: Rockwell International Corporation
Ada Validation Facility: ASD/SCEL, Wright-Patterson AFB, OH
ACVC Version: 1.11
Ada Implementation:
 Compiler Name: DDC-Based Ada/CAPS Compiler, Version 6.0
 Host Computer System: VAXstation 3100 Model 30 VMS 5.4
 Target Computer System: CAPS/AAMP2 (bare machine)

Customer's Declaration

I, the undersigned, representing Rockwell International Corporation, declare that Rockwell has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation listed in this declaration. I declare that Rockwell International Corporation is the owner of the above implementation and the certificates shall be awarded in the name of the owner's corporate name.

C. E. Kress

Date: 3/6/91

C. E. Kress, Manager of Processor Technology Department
Rockwell International Corporation
400 Collins Rd MS 124-211
Cedar Rapids, Iowa 52498

TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	
1.1	USE OF THIS VALIDATION SUMMARY REPORT	1-1
1.2	REFERENCES	1-2
1.3	ACVC TEST CLASSES	1-2
1.4	DEFINITION OF TERMS	1-3
CHAPTER 2	IMPLEMENTATION DEPENDENCIES	
2.1	WITHDRAWN TESTS	2-1
2.2	INAPPLICABLE TESTS	2-1
2.3	TEST MODIFICATIONS	2-3
CHAPTER 3	PROCESSING INFORMATION	
3.1	TESTING ENVIRONMENT	3-1
3.2	SUMMARY OF TEST RESULTS	3-1
3.3	TEST EXECUTION	3-2
APPENDIX A	MACRO PARAMETERS	
APPENDIX B	COMPILATION SYSTEM OPTIONS	
APPENDIX C	APPENDIX F OF THE Ada STANDARD	

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

INTRODUCTION

1.2 REFERENCES

- [Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
- [Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.
- [UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

INTRODUCTION

Conformity	Fulfillment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 25 February 1991.

E28005C	B28006C	C34006D	C35508I	C35508J	C35508M
C35508N	C35702A	B41308B	C43004A	C45114A	C45346A
C45612A	C45612B	C45612C	C45651A	C46022A	B49008A
A74006A	C74308A	B83022B	B83022H	B83025B	B83025D
C83026A	B83026B	C83041A	B85001L	C86001F	C94021A
C97116A	C98003B	BA2011A	CB7001A	CB7001B	CB7004A
CC1223A	BC1226A	CC1226B	BC3009B	BD1B02B	BD1B06A
AD1B08A	BD2A02A	CD2A21E	CD2A23E	CD2A32A	CD2A41A
CD2A41E	CD2A87A	CD2B15C	BD3006A	BD4008A	CD4022A
CD4022D	CD4024B	CD4024C	CD4024D	CD4031A	CD4051D
CD5111A	CD7004C	ED7005D	CD7005E	AD7006A	CD7006E
AD7201A	AD7201E	CD7204B	AD7206A	BD8002A	BD8004C
CD9005A	CD9005B	CDA201E	CE2107I	CE2117A	CE2117B
CE2119B	CE2205B	CE2405A	CE3111C	CE3116A	CE3118A
CE3411B	CE3412B	CE3607B	CE3607C	CE3607D	CE3812A
CE3814A	CE3902B				

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by the ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

IMPLEMENTATION DEPENDENCIES

The following 285 tests have floating-point type declarations requiring more digits than `SYSTEM.MAX_DIGITS`:

C24113F..Y (20 tests)	C35705F..Y (20 tests)
C35706F..Y (20 tests)	C35707F..Y (20 tests)
C35708F..Y (20 tests)	C35802F..Z (21 tests)
C45241F..Y (20 tests)	C45321F..Y (20 tests)
C45421F..Y (20 tests)	C45521F..Z (21 tests)
C45524F..Z (21 tests)	C45621F..Z (21 tests)
C45641F..Y (20 tests)	C46012F..Z (21 tests)

The following 21 tests check for the predefined type `SHORT_INTEGER`:

C35404B	B36105C	C45231B	C45304B	C45411B
C45412B	C45502B	C45503B	C45504B	C45504E
C45611B	C45613B	C45614B	C45631B	C45632B
B52004E	C55B07B	B55B09D	B86001V	C86006D
CD7101E				

C35404D, C45231D, B86001X, C86006E, and CD7101G check for a predefined integer type with a name other than `INTEGER`, `LONG_INTEGER`, or `SHORT_INTEGER`.

C35713B, C45423B, B86001T, and C86006H check for the predefined type `SHORT_FLOAT`.

C35713D and B86001Z check for a predefined floating-point type with a name other than `FLOAT`, `LONG_FLOAT`, or `SHORT_FLOAT`.

C45531M..P (4 tests) and C45532M..P (4 tests) check fixed-point operations for types that require a `SYSTEM.MAX_MANTISSA` of 47 or greater.

C45624A..B (2 tests) check that the proper exception is raised if `MACHINE_OVERFLOW` is `FALSE` for floating point types; for this implementation, `MACHINE_OVERFLOW` is `TRUE`.

C46013B, C46031B, C46033B, and C46034B contain 'SMALL representation clauses that are not integers or reciprocals of integers.

B86001Y checks for a predefined fixed-point type other than `DURATION`.

C96005B checks for values of type `DURATION'BASE` that are outside the range of `DURATION`. There are no such values for this implementation.

CA2009C, CA2009F, BC3204C, and BC3205D check whether a generic unit can be instantiated `BEFORE` its generic body (and any of its subunits) is compiled. This compiler rejects the generic body at compilation. (See section 2.3.)

CD1009C uses a representation clause specifying a non-default size for a floating-point type.

IMPLEMENTATION DEPENDENCIES

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use representation clauses specifying non-default sizes for access types.

The following 264 tests check for sequential, text, and direct access files:

CE2102A..C (3)	CE2102G..H (2)	CE2102K	CE2102N..Y (12)
CE2103C..D (2)	CE2104A..D (4)	CE2105A..B (2)	CE2106A..B (2)
CE2107A..H (8)	CE2107L	CE2108A..H (8)	CE2109A..C (3)
CE2110A..D (4)	CE2111A..I (9)	CE2115A..B (2)	CE2120A..B (2)
CE2201A..C (3)	EE2201D..E (2)	CE2201F..N (9)	CE2203A
CE2204A..D (4)	CE2205A	CE2206A	CE2208B
CE2401A..C (3)	EE2401D	CE2401E..F (2)	EE2401G
CE2401H..L (5)	CE2403A	CE2404A..B (2)	CE2405B
CE2406A	CE2407A..B (2)	CE2408A..B (2)	CE2409A..B (2)
CE2410A..B (2)	CE2411A	CE3102A..C (3)	CE3102F..H (3)
CE3102J..K (2)	CE3103A	CE3104A..C (3)	CE3106A..B (2)
CE3107B	CE3108A..B (2)	CE3109A	CE3110A
CE3111A..B (2)	CE3111D..E (2)	CE3112A..D (4)	CE3114A..B (2)
CE3115A	CE3119A	EE3203A	EE3204A
CE3207A	CE3208A	CE3301A	EE3301B
CE3302A	CE3304A	CE3305A	CE3401A
CE3402A	EE3402B	CE3402C..D (2)	CE3403A..C (3)
CE3403E..F (2)	CE3404B..D (3)	CE3405A	EE3405B
CE3405C..D (2)	CE3406A..D (4)	CE3407A..C (3)	CE3408A..C (3)
CE3409A	CE3409C..E (3)	EE3409F	CE3410A
CE3410C..E (3)	EE3410F	CE3411A	CE3411C
CE3412A	EE3412C	CE3413A..C (3)	CE3414A
CE3602A..D (4)	CE3603A	CE3604A..B (2)	CE3605A..E (5)
CE3606A..B (2)	CE3704A..F (6)	CE3704M..O (3)	CE3705A..E (5)
CE3705D	CE3706F..G (2)	CE3804A..P (16)	CE3805A..B (2)
CE3806A..B (2)	CE3806D..E (2)	CE3806G..H (2)	CE3904A..B (2)
CE3905A..C (3)	CE3905L	CE3906A..C (3)	CE3906E..F (2)

CE2103A, CE2103B, and CE3107A raise the exception `USE_ERROR` because this implementation does not support external file `CREATE` and `OPEN` operations. (See Section 2.3)

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 19 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B33301B	B55A01A	B83E01C	B83E01D	B83E01E	BA1001A
BA1101B	BC1109A	BC1109C	BC1109D		

IMPLEMENTATION DEPENDENCIES

C83030C and C86007A were graded passed by Test Modification as directed by the AVO. These tests were modified by inserting "PRAGMA ELABORATE (REPORT);" before the package declarations at lines 13 and 11, respectively. Without the pragma, the packages may be elaborated prior to package Report's body, and thus the packages' calls to function REPORT_IDENT_INT at lines 14 and 13, respectively, will raise PROGRAM_ERROR.

CA2009C, CA2009F, BC3204C, and BC3205D were graded inapplicable by Evaluation Modification as directed by the AVO. These tests contain instantiations of a generic unit prior to the compilation of that unit's body; as allowed by AI-257, this implementation requires that the bodies of a generic unit be in the same compilation if instantiations of that unit precede the bodies. The compilation of the generic unit bodies was rejected.

CE2103A, CE2103B, and CE3107A were graded inapplicable by Evaluation Modification as directed by the AVO. The tests abort with an unhandled exception when USE_ERROR is raised on the attempt to create an external file. This is acceptable behavior because this implementation does not support external files (cf. AI-00332).

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For a point of contact for technical information about this Ada implementation system, see:

Dan Lyttle
Rockwell International Corporation
MS 124-211
400 Collins Rd NE
Cedar Rapids IA 52498

For a point of contact for sales information about this Ada implementation system, see:

Charlie Kress
Rockwell International Corporation
MS 124-211
400 Collins Rd NE
Cedar Rapids IA 52498

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

PROCESSING INFORMATION

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

a) Total Number of Applicable Tests	3468
b) Total Number of Withdrawn Tests	92
c) Processed Inapplicable Tests	61
d) Non-Processed I/O Tests	264
e) Non-Processed Floating-Point Precision Tests	285
f) Total Number of Inapplicable Tests	610
g) Total Number of Tests for ACVC 1.11	4170

The above number of I/O tests were not processed because this implementation does not support a file system. The above number of floating-point tests were not processed because they used floating-point precision exceeding that supported by the implementation. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors.

3.3 TEST EXECUTION

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the magnetic tape were loaded directly onto a VAX 8650 and then copied via Ethernet to the host computer.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

The tests were compiled and linked on the host computer system, as appropriate. The executable images were transferred to the target computer system and run. The results were uploaded from the PC which controls the target to the host computer system.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were:

Option/Switch	Effect
-----	-----
/DEBUG	Enable generation of Debugger Symbol

PROCESSING INFORMATION

Table files.

/OPTIMIZE

Various target-independent optimizations
and a peephole optimizer are enabled.

Test output, compiler and linker listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for \$MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
\$BIG_ID1	(1..V-1 => 'A', V => '1')
\$BIG_ID2	(1..V-1 => 'A', V => '2')
\$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
\$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
\$BIG_INT_LIT	(1..V-3 => '0') & "298"
\$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
\$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
\$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
\$BLANKS	(1..V-20 => ' ')
\$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
\$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
\$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

MACRO PARAMETERS

The following table lists all of the other macro parameters and their respective values:

Macro Parameter	Macro Value
\$MAX_IN_LEN	126
\$ACC_SIZE	32
\$ALIGNMENT	1
\$COUNT_LAST	2_147_483_647
\$DEFAULT_MEM_SIZE	16_777_216
\$DEFAULT_STOR_UNIT	16
\$DEFAULT_SYS_NAME	AAMP2
\$DELTA_DOC	2#1.0#E-31
\$ENTRY_ADDRESS	16#40#
\$ENTRY_ADDRESS1	16#80#
\$ENTRY_ADDRESS2	16#100#
\$FIELD_LAST	36
\$FILE_TERMINATOR	ASCII.SUB
\$FIXED_NAME	NO_SUCH_FIXED_TYPE
\$FLOAT_NAME	NO_SUCH_FLOAT_TYPE
\$FORM_STRING	""
\$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
\$GREATER_THAN_DURATION	131_071.0
\$GREATER_THAN_DURATION BASE LAST	131_073.0
\$GREATER_THAN_FLOAT_BASE LAST	1.80141E+38
\$GREATER_THAN_FLOAT_SAFE LARGE	1.0E+308

MACRO PARAMETERS

\$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE
 1.0E+308

 \$HIGH_PRIORITY 254

 \$ILLEGAL_EXTERNAL_FILE_NAME1
 NO_SUCH_FILE_NAME_1

 \$ILLEGAL_EXTERNAL_FILE_NAME2
 NO_SUCH_FILE_NAME_2

 \$INAPPROPRIATE_LINE_LENGTH
 -1

 \$INAPPROPRIATE_PAGE_LENGTH
 -1

 \$INCLUDE_PRAGMA1 PRAGMA INCLUDE ("A28006D1.TST")

 \$INCLUDE_PRAGMA2 PRAGMA INCLUDE ("B28006F1.TST")

 \$INTEGER_FIRST -32768

 \$INTEGER_LAST 32767

 \$INTEGER_LAST_PLUS_1 32768

 \$INTERFACE_LANGUAGE ASSEMBLY

 \$LESS_THAN_DURATION -131_071.0

 \$LESS_THAN_DURATION_BASE_FIRST
 -131_073.0

 \$LINE_TERMINATOR ASCII.LF

 \$LOW_PRIORITY 1

 \$MACHINE_CODE_STATEMENT
 MACHINE_CODE.CODE'("NOP; ");

 \$MACHINE_CODE_TYPE CODE

 \$MANTISSA_DOC 31

 \$MAX_DIGITS 9

 \$MAX_INT 2147483647

 \$MAX_INT_PLUS_1 2147483648

 \$MIN_INT -2147483648

MACRO PARAMETERS

\$NAME	NO_SUCH_TYPE_AVAILABLE
\$NAME_LIST	AAMP1, AAMP2
\$NAME_SPECIFICATION1	[INAPPLICABLE]X2120A
\$NAME_SPECIFICATION2	[INAPPLICABLE]X2120B
\$NAME_SPECIFICATION3	[INAPPLICABLE]X2120C
\$NEG_BASED_INT	16#FOOOOOOE#
\$NEW_MEM_SIZE	16_777_216
\$NEW_STOR_UNIT	16
\$NEW_SYS_NAME	AAMP2
\$PAGE_TERMINATOR	ASCII.FF
\$RECORD_DEFINITION	RECORD INSTR: STRING (1..80); END RECORD;
\$RECORD_NAME	CODE
\$TASK_SIZE	32
\$TASK_STORAGE_SIZE	256
\$TICK	0.0001
\$VARIABLE_ADDRESS	16#02_F100#
\$VARIABLE_ADDRESS1	16#02_F200#
\$VARIABLE_ADDRESS2	16#02_F300#
\$YOUR_PRAGMA	EXPORT

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to linker documentation and not to this report.

COMPILATION SYSTEM OPTIONS

The command to invoke the Ada Compiler has the following syntax:

```
ADAC { command-qualifier } source-file-spec
```

Examples:

```
ADAC/LIST TESTPROG
```

```
ADAC/LIBRARY=MY_LIBRARY TEST
```

Parameter:

- o source-file-spec

The source-file-spec specifies the text file containing the source text of the compilation units to be compiled. If this parameter is omitted, the user will be prompted for it. If the file type is omitted, .ADA is assumed by default. No wild card characters are allowed in the file specification. The "name" portion of the file-spec is limited to 32 characters, not counting any directory, file type, or version specifications.

Qualifiers:

Default values exist for all qualifiers as indicated below. Any qualifier name may be abbreviated (characters omitted from the right) as long as no ambiguity arises.

```
/CHECK (default)
/NOCHECK [ = ( option-list ) ]
```

Example:

```
ADAC/NOCHECK=(RANGE,INDEX) TESTPROG
```

This qualifier directs the compiler to omit run-time checks from the generated object code. By default, or when /CHECK is specified, all run-time checks required by the language are generated.

When the /NOCHECK qualifier is used, the effect is the same as if the unit(s) being compiled had included a Pragma SUPPRESS for each check in the option-list, or for all checks if the option-list is omitted.

The inclusion of Pragma SUPPRESS in the unit being compiled takes precedence over the use of /CHECK as a compiler option. The result of using either Pragma SUPPRESS or the /NOCHECK option is a non-standard Ada program.

COMPILATION SYSTEM OPTIONS

The following checks may be individually suppressed:

ACCESS	- Check for access value being non-null.
DISCRIMINANT	- Check for record discriminant value.
ELABORATION	- Check for subprogram being elaborated.
INDEX	- Check for array index value in range.
LENGTH	- Check for array length compatibility.
RANGE	- Check for value within subtype range.
STORAGE	- Has no effect.
ALL	- All of the above checks suppressed.

```
/CMS GENERATIONS [ = "quoted-string" ]  
/NOCMS_GENERATIONS (default)
```

This qualifier is used to pass configuration management information through the compiler system to the load module created from the Ada program. The version identification of each source file used in construction of the load module can be displayed using the CTRACE tool.

When the /CMS qualifier is used by itself, the source file is scanned to extract the history notes stored in the file by the DEC Code Management System (CMS).

The CMS qualifier /HISTORY="-- B" must be in effect when the source file is fetched, causing CMS to store the information in the file for the compiler system to use.

If an optional value is specified, for example, ADAC/CMS="2.1", this value is used instead. This allows a configuration management system other than DEC CMS to be used, e.g., the Rational Target Build Facility.

See the Ada CAPS Object/Load Module Information Trace (CTRACE) User's Guide for further information.

```
/CONFIGURATION_FILE = file-spec  
/CONFIGURATION_FILE = ADACS_CONFIG (default)
```

This qualifier specifies the configuration file to be used by the compiler in the current compilation. If the qualifier is omitted, the configuration file designated by the logical name ADACS_CONFIG is used. Section 3.3.2.1 contains a description of the configuration file.

```
/DEBUG (default)  
/NODEBUG
```

When /NODEBUG is active, the debugger file (having extension .DST) will not be created. The default is to create this file.

```
/KEEP = ( [ ASM, LST ] )
```

COMPILATION SYSTEM OPTIONS

This qualifier prevents the deletion of the assembly source (.ASM) and/or assembly listing (.LST) files created during compilation. The default is to delete the .ASM and .LST files. No corresponding /NOKEEP qualifier is allowed.

```
/LIBRARY = file-spec  
/LIBRARY = ADACS_LIBRARY          (default)
```

This qualifier specifies the current sublibrary and thereby also the current program library (cf. section 2).

If the qualifier is omitted, the sublibrary designated by the logical name ADACS_LIBRARY is used as the current sublibrary (cf. section 2.1). Section 3.3.4 describes how the Ada compiler uses the library.

```
/LIST  
/NOLIST                          (default)
```

When this qualifier is given, the source listing is written on the list file. Section 3.3.3.1 contains a description of the source listing.

If /NOLIST is active, no source listing is produced, regardless of any LIST pragmas in the program or any diagnostic messages produced.

```
/MACASM = ( [ CROSS_REFERENCE, STATISTICS, TIMING ] )  
/MACASM = (NOCROSS, _NOSTAT, NOTIMING)          (default)
```

Inclusion of any of these parameters causes the macro assembler to include cross-reference information, assembler execution statistics, or instruction timing information respectively, in its generated list file. The source listing information is always included in the macro assembler list file. By default, these sections are not included in the macro assembler list file.

Note: The /MACASM qualifier has no visible effect unless the /KEEP = LST qualifier is also present.

```
/OBJECT                          (default)  
/NOOBJECT
```

This qualifier indicates whether object code will be generated if the compilation is successful. Compilation is considerably faster when /NOOBJECT is specified, but the program cannot be linked if a required object file is missing.

COMPILATION SYSTEM OPTIONS

/OPTIMIZE = (list-of-optimizations)
/NOOPTIMIZE (default)

This qualifier specifies whether any optimizations should be performed on the generated code. Specifying /OPTIMIZE without a list-of-optimizations will cause all optimizations to be performed. The default is not to perform any optimizations.

The following optimizations may be individually selected:

- CHECK - The compiler will eliminate unneeded run-time checks. This option should NOT be used if /NOCHECK is used.
- CSE - Common subexpression elimination is performed.
- PEEP - Peep-hole optimization is performed on the generated code before object code is produced. This involves scanning the assembly source for the occurrence of certain patterns of code which can be replaced by simpler, equivalent patterns.
- REORDERING - The compiler will reorder an aggregate with named component association into an aggregate with positional association, which may cause it to be placed in ROM instead of generating initialization code.
- STACK_HEIGHT - The use of temporary variables in expression evaluation is minimized.
- ALL - Causes all optimizations to be performed.

Example:

ADAC/OPTIMIZE=(PEEP,CSE) TESTPROG

/PRAGMA=UNIVERSAL_DATA

This qualifier is equivalent to placing "Pragma UNIVERSAL_DATA;" in the declarative part of the unit(s) being compiled. This allows the user to overcome the restriction that all stacks and volatile data be located in a single 64K word memory sector when the program is linked. It is only allowed when compiling a library package spec or body. See Appendix F of the ADACS Reference Manual for further information on this implementation-dependent pragma.

COMPILATION SYSTEM OPTIONS

/PROGRESS
/NOPROGRESS (default)

When this qualifier is given, the compiler will output information about which phase of the compiler is currently running.

/SAVE_SOURCE (default)
/NOSAVE_SOURCE

This qualifier specifies whether the source text is stored in the program library. If the source file contains several compilation units, the source text for each compilation unit is stored in the program library.

The source texts stored in the program library can be extracted using the PLU command TYPE. Using the /NOSAVE qualifier will prevent automatic recompilation using the Ada Recompiler.

/XREF
/NOXREF (default)

This qualifier is used to generate a cross-reference listing. If the /XREF qualifier is given and no severe or fatal errors are found during compilation, the cross-reference listing is written on the list file.

The command to invoke the Ada Linker has the following syntax:

```
ADAC/LINK { command-qualifier } unit name
          [ recompilation-spec ]

<recompilation-spec> ::= <unit-spec> { + <unit-spec> }
                      | <unit-spec> { , <unit-spec> }

<unit-spec>          ::= <unit-name>
                      | <unit-name> / SPECIFICATION
                      | <unit-name> / BODY
```

Parameters:

- o unit-name

If a link is requested (no recompilation-spec was given), unit-name specifies a main program which must be a library unit of the current program library, but not necessarily of the current sublibrary. The main program must be a parameterless procedure. The linker does not check that the main procedure has no parameters, but execution of a main program with parameters is undefined.

Note that unit-name is the Ada procedure name, not a file name.

If an examination of the consequences of recompilations is requested (a recompilation-spec is given), unit-name specifies a set of program library units whose consistency will be checked as if the hypothetical recompilations had just occurred. The recompilation-spec may include wildcard characters, which will be interpreted according to VAX/VMS rules for wildcard characters.

The following applies to the different kinds of unit-names:

- If unit-name does not contain wildcard characters, it designates the visible unit having the specified name. The designated unit must be a parameterless procedure.
- If unit-name contains wildcard characters, it designates all library units in the current sublibrary with names matching the specified unit-name. All types of library units may be designated.

o recompilation-spec

If this parameter is given, the linker will analyze the consequences (for the unit(s) named by the unit-name parameter) of hypothetical recompilations of the unit or units given by the list.

A recompilation-spec is a sequence of unit-specs, separated by plus or comma characters. Each unit-spec is a unit-name, possibly with a /SPECIFICATION or /BODY qualifier. If a unit-name does not have this qualifier, /SPECIFICATION is the default.

Qualifiers:

```
/CLINK = ( [ FULL, LIST=file-spec, MAP, CROSS REFERENCE,
           STATISTICS, SEGMENT=name, RAM, ROM ] )
/NOCLINK
/CLINK = (FULL)                                (default)
```

This qualifier is used to pass qualifiers to CLINK, the CAPS Linker, which is part of the linker system. All of these qualifiers may be negated except for SEGMENT. The use of LIST, MAP, CROSS REFERENCE, or STATISTICS implies NOFULL.

To omit selected parts of the program from the load module (for example, when the Executive is already in ROM), the SEGMENT qualifier is recommended instead of the older RAM

COMPILATION SYSTEM OPTIONS

and ROM qualifiers. See the CAPS Link Editor User's Guide for more information.

The /NOCLINK qualifier is used to prevent execution of the target linker. This is typically used when special processing is required before target linking. After the linker terminates, the .LEC file produced by the linker can be modified by the user, who then invokes the target linker (CLINK) separately.

/DEBUG (default)
/NODEBUG

The /NODEBUG qualifier is used to prevent creation of a .DMT Debugger Module Table file by the linker. By default, the debugger file is created.

/LIBRARY = file-spec
/LIBRARY = ADACS_LIBRARY (default)

This qualifier specifies the current sublibrary and thereby also the current program library. The current program library consists of the current sublibrary and its ancestor sublibraries (cf. section 2.1).

If the qualifier is omitted, the sublibrary designated by the logical name ADACS_LIBRARY is used as current sublibrary.

/LOG [= file-spec]
/NOLOG (default)

The qualifier determines whether a log file will be produced. By default, no log file is produced. If a file specification is given, this file will be used as the log file. If /LOG is specified without a file specification, a log file named main-program-name.LSF is created in the current default directory. The log file contents are described in Section 3.4.3.3.

/TARGET_LINKER_COMMANDS = file-name
/TARGET_LINKER_COMMANDS =
ADACS_SUPPORT_LIBRARY:exec_target.LEC
(default)

where "exec" is TASKING or NOTASKING depending on the /TASKING qualifier, and "target" is AAMP1 or AAMP2 depending on the target selected by the @ADACS:[TOOLS]USE command (for example, TASKING_AAMP1.LEC).

This qualifier allows the user to specify that a Link Edit Command (.LEC) file other than the default one be used as input during linking. The .LEC file is used to instruct

the linker how to construct the output module. As the linker generates an intermediate file named main-procedure-name.LEC, the user must not specify a file with this name on this qualifier. For more information, consult the CAPS Link Editor User's Guide or Application Note 4.

/TASKING (default)
/NOTASKING

The /NOTASKING qualifier is used to indicate to the linker that the program to be linked does not contain any Ada tasking constructs. The linked code for a program which uses this option will be smaller than if the option is not used. The user's main program will run in executive mode of the processor. The default is to allow tasking constructs in a program to be linked.

Examples:

ADAC/LINK/LIBRARY=MYLIB.LIB MYPROG

The linker will generate an executable load module from the program MYPROG found in the library defined by the current sublibrary MYLIB.LIB.

ADAC/LINK/LOG MYPROG EXAMPLE/SPEC, UTILITY/BODY

This will examine the consequence of recompiling the specification of EXAMPLE and the body of UTILITY. The linker will give a list of necessary compilations to keep MYPROG consistent. The program library is given by the default name ADACS_LIBRARY.

ADAC/LINK PROG_X A/SPEC + B*/SPEC + C%*/BODY

Here, the linker will examine the consistency of the program "PROG_X" in case of a recompilation of the specifications of the library unit "A" and all library units with names starting with "B", and a recompilation of the bodies of all library units with names at least 3 characters long, whose first character is "C" and whose third character is "_".

ADAC/LINK MYPROG DUMMY_UNIT_NAME

Assume no unit of the name DUMMY_UNIT_NAME exists in the current library. Then this command will examine the consequences of no recompilations; the linker simply checks the consistency of MYPROG without linking it.

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

package STANDARD is

```
...
type INTEGER is range -32_768 .. 32_767;
type LONG_INTEGER is range -2_147_483_648 .. 2_147_483_647;

type FLOAT is digits 6 range -16#0.7FFF_FF8#E32 .. 16#0.7FFF_FF8#E32;
type LONG_FLOAT is digits 9 range -16#0.7FFF_FFFF_FF8#E32 ..
                                     16#0.7FFF_FFFF_FF8#E32;

type DURATION is delta 0.0001 range -16#2_0000.0# .. 16#1_FFFF.FF8#;
...
```

end STANDARD;

APPENDIX F OF THE Ada STANDARD

```
*****
*
*           DDC-Based
*       Ada/CAPS Compiler System
*       Language Reference Manual
*
*
* Document Identification: REF_MAN
* Document Number: DADA-251
* Date: 31 January 1991
* Compiler System Version: 6.0
*
*****
```

D. W. Lyttle

Computer Support System Section
Processor Technology Department
Advanced Technology Department
Collins Government Avionics Division

Avionics Group
400 Collins Road NE
Cedar Rapids, Iowa 52498

(c) Rockwell International Corporation, 1991
All Rights Restricted

Proprietary Data. Dissemination outside Rockwell
International Corporation must be cleared through
the Patent Department.

CONTENTS

1	Introduction	2
1.1	References	2
12	Generic Units	3
12.3	Generic Instantiation	3
12.3.2	Matching Rules For Formal Private Types	3

APPENDICES

A PREDEFINED LANGUAGE ATTRIBUTES

B PREDEFINED LANGUAGE PRAGMAS

C PREDEFINED LANGUAGE ENVIRONMENT

D GLOSSARY

E SYNTAX SUMMARY

F	IMPLEMENTATION-DEPENDENT CHARACTERISTICS	
F.1	Implementation-Dependent Pragmas.	F-1
F.1.1	Pragma EXPORT.	F-1
F.1.2	Pragma IMPORT.	F-3
F.1.3	Pragma STACK SIZE.	F-4
F.1.4	Pragma UNIVERSAL DATA	F-5
F.2	Implementation-Dependent Attributes.	F-6
F.3	Specification Of The Package SYSTEM.	F-6
F.4	Representation Clauses.	F-6
F.4.1	Length Clauses.	F-6
F.4.1.1	Size Specifications.	F-6
F.4.1.2	Collection Size Specification.	F-7
F.4.1.3	Task STORAGE SIZE Specification.	F-8
F.4.1.4	SMALL Specification.	F-8
F.4.2	Enumeration Representation Clauses.	F-8
F.4.3	Record Representation Clauses.	F-8
F.4.4	Alignment Clauses.	F-10
F.4.5	Implementation-Generated Names.	F-10
F.5	Address Clause Expressions.	F-10
F.6	UNCHECKED_CONVERSION Restrictions.	F-10

F.7	I/O Package Implementation-Dependent	
	Characteristics.	F-10
F.7.1	Package SEQUENTIAL_IO.	F-11
F.7.2	Package DIRECT_IO.	F-11
F.7.3	Package TEXT_IO.	F-11
F.7.4	Package LOW_LEVEL_IO.	F-11
F.7.5	Package LOW_IO.	F-12
F.8	Other Implementation-Dependent Features.	F-12
F.8.1	Predefined Types.	F-12
F.8.1.1	Integer Types.	F-12
F.8.1.2	Floating Point Types.	F-12
F.8.1.3	Fixed Point Types.	F-14
F.8.1.4	The Type DURATION.	F-15
F.8.2	Uninitialized Variables.	F-15
F.8.3	Package MACHINE_CODE.	F-16
F.8.4	Compiler Limitations.	F-17

1 Introduction

This document is a supplement to the Ada Language Reference Manual (RM), ANSI/MIL-STD-1815A, 22-Jan-83 (Chapters 1-14). The appendices correspond to similar sections of the RM. Appendices A, B, and C contain a superset of Annexes A, B, and C of the RM, with added lines containing a vertical bar at the left, indicating changes specific to Ada/CAPS. Appendix D is a glossary of Ada/CAPS related terms and should be considered as an extension of Annex D of the RM. Appendix E contains only a reference to the RM since the syntax of Ada/CAPS is identical to that of standard Ada. Appendix F describes the implementation-dependent characteristics of the Ada/CAPS compiler system.

1.1 References

Each of the ADACS documents is available both as an .LNI file for sending to a DEC LNO3 laser printer, and as a .MEM file for sending to a standard line printer.

1. Ada Language Reference Manual, ANSI/MIL-STD-1815A, 1983. Describes the Department of Defense Military Standard Ada Language. Available from the DoD Single Stock Point, Commanding Officer, Naval Publications and Forms Center, 5801 Tabor Avenue, Philadelphia PA 19120.
2. DDC-Based Ada/CAPS Cross Compiler System User's Guide. Describes how to run the tools of the system. Available by printing the file ADACS:[DOCUMENTS]USERS_GUIDE.MEM.
3. DDC-Based Ada/CAPS Cross Compiler System Run-time Reference Manual. Describes implementation details of Ada/CAPS in the context of the target machine. Available by printing the file:
ADACS:[DOCUMENTS]RUNTIME_REFERENCE_MANUAL.MEM
4. CAPS Macro Assembler User's Guide. Contains information on how to call macros, write assembly instructions, and use the macro assembler. Available by printing the file:
ADACS:[DOCUMENTS]MACASM.MEM.
5. CAPS Link Editor User's Guide. Describes the CAPS Linker. Available by printing the file:
ADACS:[DOCUMENTS]CLINK.MEM.
6. Memory Organization and Linking Considerations. Available by printing the file:
ADACS:[DOCUMENTS]NOTE_04.MEM.

12 Generic Units

12.3 Generic Instantiation

Order of Compilation

When instantiating a generic unit, it is required that the entire unit, including body and possible subunits, is compiled before the first instantiation or, at the latest, in the same compilation. This is in accordance with the RM Chapter 10.3 (1).

12.3.2 Matching Rules For Formal Private Types

The present section describes the treatment of a generic unit with a generic formal private type, where there is some construct in the generic unit that requires that the corresponding actual type must be constrained if it is an array type or a record type with discriminants, and there exists an instantiation with such an unconstrained type (see RM Section 12.3.2.(4)).

This is considered an illegal combination. In some cases the error is detected when the instantiation is compiled; in other cases, when a constraint-requiring construct of the generic unit is compiled:

- 1) If the instantiation appears in a later compilation unit than the first constraint-requiring construct of the generic unit, the error is associated with the instantiation which is rejected by the compiler.
- 2) If the instantiation appears in the same compilation unit as the first constraint-requiring construction of the generic unit, there are two possibilities:
 - a) If there is a constraint-requiring construction of the generic unit after the instantiation, an error message appears with the instantiation.
 - b) If the instantiation appears after all constraint-requiring constructs of the generic unit in that compilation unit, an error message appears with the constraint-requiring construct, but will refer to the illegal instantiation.

- 3) The instantiation appears in an earlier compilation unit than the first constraint-requiring construction of the generic unit, which in that case will appear in the generic body or a subunit. If the instantiation has been accepted, the instantiation will correspond to the generic declaration only, and not include the body. Nevertheless, if the generic unit and the instantiation are located in the same sublibrary, then the compiler will consider it an error. An error message will be issued with the constraint-requiring construct and will refer to the illegal instantiation. The unit containing the instantiation is not changed, however, and will not be marked as invalid.

APPENDIX A

PREDEFINED LANGUAGE ATTRIBUTES

This annex summarizes the definitions given elsewhere of the predefined language attributes.

- | | |
|-----------|---|
| P'ADDRESS | <p>For a prefix P that denotes an object, a program unit, a label, or an entry:</p> <p>Yields the address of the first of the storage units allocated to P. For a subprogram, package, task unit, or label, this value refers to the machine code associated with the corresponding body or statement. For an entry for which an address clause has been given, the value refers to the corresponding hardware interrupt. The value of this attribute is of the type ADDRESS defined in the package SYSTEM (see F.3).</p> |
| P'FIRST | <p>To be consistent with the machine architecture, a word address is returned for a data object, but a byte address is returned for code items.</p> |
| P'AFT | <p>For a prefix P that denotes a fixed point subtype:</p> <p>Yields the number of decimal digits needed after the point to accommodate the precision of the subtype P, unless the delta of the subtype P is greater than 0.1, in which case the attribute yields the value one. (P'AFT is the smallest positive integer N for which $(10^{**}N) * P'DELTA$ is greater than or equal to one.) The value of this attribute is of the type universal_integer. (See 3.5.10.)</p> |
| P'BASE | <p>For a prefix P that denotes a type or subtype:</p> <p>This attribute denotes the base type of P. It is only allowed as the prefix of the name of another attribute: for example: P'BASE'FIRST. (See 3.3.3.)</p> |

PREDEFINED LANGUAGE ATTRIBUTES

Page A-2

- P'CALLABLE** For a prefix P that is appropriate for a task type:
- Yields the value FALSE when the execution of the task P is either completed or terminated, or when the task is abnormal; yields the value TRUE otherwise. The value of this attribute is of the predefined type BOOLEAN. (See 9.9.)
- P'CONSTRAINED** For a prefix P that denotes an object of a type with discriminants:
- Yields the value TRUE if a discriminant constraint applies to the object P, or if the object is a constant (including a formal parameter or generic formal parameter of mode in); yields the value FALSE otherwise. If P is a generic formal parameter of mode in out, or if P is a formal parameter of mode in out or out and the type mark given in the corresponding parameter specification denotes an unconstrained type with discriminants, then the value of this attribute is obtained from that of the corresponding actual parameter. The value of this attribute is of the predefined type BOOLEAN. (See 3.7.4.)
- P'CONSTRAINED** For a prefix P that denotes a private type or subtype:
- Yields the value FALSE if P denotes an unconstrained nonformal private type with discriminants; also yields the value FALSE if P denotes a generic formal private type and the associated actual subtype is either an unconstrained type with discriminants or an unconstrained array type; yields the value TRUE otherwise. The value of this attribute is of the predefined type BOOLEAN. (See 7.4.2.)
- P'COUNT** For a prefix P that denotes an entry of a task unit:
- Yields the number of entry calls presently queued on the entry (if the attribute is evaluated within an accept statement for the entry P, the count does not include the calling task). The value of this attribute is of the type `universal_integer`. (See 9.9.)

PREDEFINED LANGUAGE ATTRIBUTES

Page A-3

- P'DELTA** For a prefix *P* that denotes a fixed point subtype:
- Yields the value of the delta specified in the fixed accuracy definition for the subtype *P*. The value of this attribute is of the type `universal_real`. (See 3.5.10.)
- P'DIGITS** For a prefix *P* that denotes a floating point subtype:
- Yields the number of decimal digits in the decimal mantissa of model numbers of the subtype *P*. (This attribute yields the number *D* of section 3.5.7.) The value of this attribute is of the type `universal_integer`. (See 3.5.8.)
- P'EMAX** For a prefix *P* that denotes a floating point subtype:
- Yields the largest exponent value in the binary canonical form of model numbers of the subtype *P*. (This attribute yields the product $4*B$ of section 3.5.7.) The value of this attribute is of the type `universal_integer`. (See 3.5.8.)
- P'EPSILON** For a prefix *P* that denotes a floating point subtype:
- Yields the absolute value of the difference between the model number 1.0 and the next model number above, for the subtype *P*. The value of this attribute is of the type `universal_real`. (See 3.5.8.)
- P'FIRST** For a prefix *P* that denotes a scalar type, or a subtype of a scalar type:
- Yields the lower bound of *P*. The value of this attribute has the same type as *P*. (See 3.5.)
- P'FIRST** For a prefix *P* that is appropriate for an array type, or that denotes a constrained array subtype:
- Yields the lower bound of the first index range. The value of this attribute has the same type as this lower bound. (See 3.6.2 and 3.8.2.)

PREDEFINED LANGUAGE ATTRIBUTES

Page A-4

- P'FIRST(N)** For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:
- Yields the lower bound of the N-th index range. The value of this attribute has the same type as this lower bound. The argument N must be a static expression of type `universal_integer`. The value of N must be positive (nonzero) and no greater than the dimensionality of the array. (See 3.6.2 and 3.8.2.)
- P'FIRST_BIT** For a prefix P that denotes a component of a record object:
- Yields the offset, from the start of the first of the storage units occupied by the component, of the first bit occupied by the component. This offset is measured in bits. The value of this attribute is of the type `universal_integer`. (See 13.7.2.)
- P'FORE** For a prefix P that denotes a fixed point subtype:
- Yields the minimum number of characters needed for the integer part of the decimal representation of any value of the subtype P, assuming that the representation does not include an exponent, but includes a one-character prefix that is either a minus sign or a space. (This minimum number does not include superfluous zeros or underlines, and is at least two.) The value of this attribute is of the type `universal_integer`. (See 3.5.10.)
- P'IMAGE** For a prefix P that denotes a discrete type or subtype:
- This attribute is a function with a single parameter. The actual parameter X must be a value of the base type of P. The result type is the predefined type `STRING`. The result is the image of the value of X, that is, a sequence of characters representing the value in display form. The image of an integer value is the corresponding decimal literal; without underlines, leading zeros, exponent, or trailing spaces; but with a one-character prefix that is either a minus sign or a space.

The image of an enumeration value is either the corresponding identifier in upper case or the corresponding character literal (including the two apostrophes); neither leading nor trailing spaces are included. The image of a character other than a graphic character is implementation-defined. (See 3.5.5.)

P' LARGE

For a prefix P that denotes a real subtype:

The attribute yields the largest positive model number of the subtype P. The value of this attribute is of the type `universal_real`. (See 3.5.8 and 3.5.10.)

P' LAST

For a prefix P that denotes a scalar type, or a subtype of a scalar type:

Yields the upper bound of P. The value of this attribute has the same type as P. (See 3.5.)

P' LAST

For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:

Yields the upper bound of the first index range. The value of this attribute has the same type as this upper bound. (See 3.6.2 and 3.8.2.)

P' LAST(N)

For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:

Yields the upper bound of the N-th index range. The value of this attribute has the same type as this upper bound. The argument N must be a static expression of type `universal_integer`. The value of N must be positive (nonzero) and no greater than the dimensionality of the array. (See 3.6.2 and 3.8.2.)

P' LAST_BIT

For a prefix P that denotes a component of a record object:

Yields the offset, from the start of the first of the storage units occupied by the component, of the last bit occupied by the component. This offset is measured in bits.

PREDEFINED LANGUAGE ATTRIBUTES

Page A-6

	The value of this attribute is of the type <code>universal_integer</code> . (See 13.7.2.)
<code>P'LENGTH</code>	<p>For a prefix <code>P</code> that is appropriate for an array type, or that denotes a constrained array subtype:</p> <p>Yields the number of values of the first index range (zero for a null range). The value of this attribute is of the type <code>universal_integer</code>. (See 3.6.2.)</p>
<code>P'LENGTH(N)</code>	<p>For a prefix <code>P</code> that is appropriate for an array type, or that denotes a constrained array subtype:</p> <p>Yields the number of values of the <code>N</code>-th index range (zero for a null range). The value of this attribute is of the type <code>universal_integer</code>. The argument <code>N</code> must be a static expression of type <code>universal_integer</code>. The value of <code>N</code> must be positive (nonzero) and no greater than the dimensionality of the array. (See 3.6.2 and 3.8.2.)</p>
<code>P'MACHINE_EMAX</code>	<p>For a prefix <code>P</code> that denotes a floating point type or subtype:</p> <p>Yields the largest value of exponent for the machine representation of the base type of <code>P</code>. The value of this attribute is of the type <code>universal_integer</code>. (See 13.7.3.)</p> <p>The value 127 is returned for <code>FLOAT</code> or <code>LONG_FLOAT</code>.</p>
<code>P'MACHINE_EMIN</code>	<p>For a prefix <code>P</code> that denotes a floating point type or subtype:</p> <p>Yields the smallest (most negative) value of exponent for the machine representation of the base type of <code>P</code>. The value of this attribute is of the type <code>universal_integer</code>. (See 13.7.3.)</p> <p>The value -127 is returned for <code>FLOAT</code> or <code>LONG_FLOAT</code>.</p>
<code>P'MACHINE_MANTISSA</code>	For a prefix <code>P</code> that denotes a floating point type or subtype:

Yields the number of digits in the mantissa for the machine representation of the base type of P (the digits are extended digits in the range 0 to P'MACHINE RADIX - 1). The value of this attribute is of the type universal_integer. (See 13.7.3.)

The value 24 is returned for FLOAT, or 40 for LONG_FLOAT.

P'MACHINE_OVERFLOW For a prefix P that denotes a real type or subtype:

Yields the value TRUE if every predefined operation on values of the base type of P either provides a correct result, or raises the exception NUMERIC_ERROR in overflow situations; yields the value FALSE otherwise. The value of this attribute is of the predefined type BOOLEAN. (See 13.7.3.)

The value TRUE is returned for a floating-point or fixed-point type.

P'MACHINE_RADIX For a prefix P that denotes a floating point type or subtype:

Yields the value of the radix used by the machine representation of the base type of P. The value of this attribute is of the type universal_integer. (See 13.7.3.)

The value 2 is returned for FLOAT or LONG_FLOAT.

P'MACHINE_ROUNDS For a prefix P that denotes a real type or subtype:

Yields the value TRUE if every predefined arithmetic operation on values of the base type of P either returns an exact result or performs rounding; yields the value FALSE otherwise. The value of this attribute is of the predefined type BOOLEAN. (See 13.7.3.)

The value TRUE is returned for a floating-point type, or FALSE for a fixed-point type.

PREDEFINED LANGUAGE ATTRIBUTES

Page A-8

- P'MANTISSA** For a prefix P that denotes a real subtype:
- Yields the number of binary digits in the binary mantissa of model numbers of the subtype P. (This attribute yields the number B of section 3.5.7 for a floating point type, or of section 3.5.9 for a fixed point type.) The value of this attribute is of the type `universal_integer`. (See 3.5.8 and 3.5.10.)
- P'POS** For a prefix P that denotes a discrete type or subtype:
- This attribute is a function with a single parameter. The actual parameter X must be a value of the base type of P. The result type is the type `universal_integer`. The result is the position number of the value of the actual parameter. (See 3.5.5.)
- P'POSITION** For a prefix P that denotes a component of a record object:
- Yields the offset, from the start of the first storage unit occupied by the record, of the first of the storage units occupied by the component. This offset is measured in storage units. The value of this attribute is of the type `universal_integer`. (See 13.7.2.)
- P'PRED** For a prefix P that denotes a discrete type or subtype:
- This attribute is a function with a single parameter. The actual parameter X must be a value of the base type of P. The result type is the base type of P. The result is the value whose position number is one less than that of X. The exception `CONSTRAINT_ERROR` is raised if X equals `P'BASE'FIRST`. (See 3.5.5.)

PREDEFINED LANGUAGE ATTRIBUTES

Page A-9

P'RANGE	<p>For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:</p> <p>Yields the first index range of P; that is, the range P'FIRST .. P'LAST. (See 3.6.2.)</p>
P'RANGE(N)	<p>For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:</p> <p>Yields the N-th index range of P, that is, the range P'FIRST(N) .. P'LAST(N). (See 3.6.2.)</p>
P'SAFE_EMAX	<p>For a prefix P that denotes a floating point type or subtype:</p> <p>Yields the largest exponent value in the binary canonical form of safe numbers of the base type of P. (This attribute yields the number E of section 3.5.7.) The value of this attribute is of the type universal_integer. (See 3.5.8.)</p>
P'SAFE_LARGE	<p>For a prefix P that denotes a real type or subtype:</p> <p>Yields the largest positive safe number of the base type of P. The value of this attribute is of the type universal_real. (See 3.5.8 and 3.5.10.)</p>
P'SAFE_SMALL	<p>For a prefix P that denotes a real type or subtype:</p> <p>Yields the smallest positive (nonzero) safe number of the base type of P. The value of this attribute is of the type universal_real. (See 3.5.8 and 3.5.10.)</p>
P'SIZE	<p>For a prefix P that denotes an object:</p> <p>Yields the number of bits allocated to hold the object. The value of this attribute is of the type universal_integer. (See 13.7.2.)</p>
P'SIZE	<p>For a prefix P that denotes any type or subtype:</p>

PREDEFINED LANGUAGE ATTRIBUTES

Page A-10

Yields the minimum number of bits that is needed by the implementation to hold any possible object of the type or subtype P. The value of this attribute is of the type `universal_integer`. (See 13.7.2.)

P'SMALL

For a prefix P that denotes a real subtype:

Yields the smallest positive (nonzero) model number of the subtype P. The value of this attribute is of the type `universal_real`. (See 3.5.8 and 3.5.10.)

P'STORAGE_SIZE

For a prefix P that denotes an access type or subtype:

Yields the total number of storage units reserved for the collection associated with the base type of P. The value of this attribute is of the type `universal_integer`. (See 13.7.2.)

P'STORAGE_SIZE

For a prefix P that denotes a task type or a task object:

Yields the number of storage units reserved for each activation of a task of the type P or for the activation of the task object P. The value of this attribute is of the type `universal_integer`. (See 13.7.2.)

Yields the number of storage units (words) reserved for the data stack of a task of type P or of the task object P.

P'SUCC

For a prefix P that denotes a discrete type or subtype:

This attribute is a function with a single parameter. The actual parameter X must be a value of the base type of P. The result type is the base type of P. The result is the value whose position number is one greater than that of X. The exception `CONSTRAINT_ERROR` is raised if X equals `P'BASE'LAST`. (See 3.5.5.)

- P'TERMINATED** For a prefix P that is appropriate for a task type:
- Yields the value TRUE if the task P is terminated; yields the value FALSE otherwise. The value of this attribute is of the predefined type BOOLEAN. (See 9.9.)
- P'VAL** For a prefix P that denotes a discrete type or subtype:
- This attribute is a special function with a single parameter X which can be of any integer type. The result type is the base type of P. The result is the value whose position number is the universal integer value corresponding to X. The exception CONSTRAINT_ERROR is raised if the universal integer value corresponding to X is not in the range P'POS(P'BASE'FIRST) .. P'POS(P'BASE'LAST). (See 3.5.5.)
- P'VALUE** For a prefix P that denotes a discrete type or subtype:
- This attribute is a function with a single parameter. The actual parameter X must be a value of the predefined type STRING. The result type is the base type of P. Any leading and any trailing spaces of the sequence of characters that corresponds to X are ignored.
- For an enumeration type, if the sequence of characters has the syntax of an enumeration literal and if this literal exists for the base type of P, the result is the corresponding enumeration value. For an integer type, if the sequence of characters has the syntax of an integer literal, with an optional single leading character that is a plus or minus sign, and if there is a corresponding value in the base type of P, the result is this value. In any other case, the exception CONSTRAINT_ERROR is raised. (See 3.5.5.)

PREDEFINED LANGUAGE ATTRIBUTES

Page A-12

P'WIDTH

For a prefix P that denotes a discrete subtype:

Yields the maximum image length over all values of the subtype P (the image is the sequence of characters returned by the attribute IMAGE). The value of this attribute is of the type universal_integer. (See 3.5.5.)

APPENDIX B

PREDEFINED LANGUAGE PRAGMAS

This annex defines the pragmas LIST, PAGE, and OPTIMIZE, and summarizes the definitions given elsewhere of the remaining language-defined pragmas.

| It also summarizes the implementation-dependent pragmas EXPORT,
| IMPORT, STACK_SIZE, and UNIVERSAL_DATA which are described in
| more detail in Appendix F.

Pragma

Meaning

CONTROLLED

Takes the simple name of an access type as the single argument. This pragma is only allowed immediately within the declarative part or package specification that contains the declaration of the access type; the declaration must occur before the pragma. This pragma is not allowed for a derived type. This pragma specifies that automatic storage reclamation must not be performed for objects designated by values of the access type, except upon leaving the innermost block statement, subprogram body, or task body that encloses the access type declaration, or after leaving the main program (see 4.8).

| This pragma has no effect, as no automatic storage
| reclamation is performed before the point allowed
| by the pragma.

ELABORATE

Takes one or more simple names denoting library units as arguments. This pragma is only allowed immediately after the context clause of a compilation unit (before the subsequent library unit or secondary unit). Each argument must be the simple name of a library unit mentioned by the context clause. This pragma specifies that the corresponding library unit body must be elaborated before the given compilation unit. If the given compilation unit is a subunit, the library unit body must be elaborated before the body of the ancestor library unit of the subunit (see 10.5).

This pragma can help the compiler generate more efficient code for calls to subprograms in the unit, as it makes an elaboration check unnecessary.

EXPORT

Takes an identifier denoting a subprogram or an object, and optionally takes a string literal (the name of a CAPS object module entry/external name) as arguments. This pragma is only allowed at the place of a declarative item and must apply to a subprogram or object declared by an earlier declarative item in the same declarative part.

The pragma must occur in the same compilation unit as the subprogram body to export a subprogram, and in the same compilation unit as the declaration to export an object. A subprogram to be exported may not be nested within anything but a library unit package specification or body. An object to be exported must be a static object; the pragma is not allowed for an access or a task object.

This pragma makes a subprogram or object visible to non-Ada parts of the system. See Appendix F for more detail.

IMPORT

Takes an internal name denoting a subprogram and optionally takes a string literal (the name of a CAPS object module entry/external name) as arguments. This pragma is only allowed at the place of a declarative item and must apply to a subprogram declared by an earlier declaration item in the same declarative part or package specification.

This pragma allows a subprogram written in another language to be called from an Ada program. See Appendix F for more detail.

INLINE

Takes one or more names as arguments; each name is either the name of a subprogram or the name of a generic subprogram. This pragma is only allowed at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation, but before any subsequent compilation unit. This pragma specifies that the subprogram bodies should be expanded inline at each call whenever possible; in the case of a generic subprogram, the pragma applies to calls of its instantiations (see 6.3.2).

PREDEFINED LANGUAGE PRAGMAS

Page B-3

- INTERFACE** Takes a language name and a subprogram name as arguments. This pragma is allowed at the place of a declarative item, and must apply in this case to a subprogram declared by an earlier declarative item of the same declarative part or package specification. This pragma is also allowed for a library unit; in this case the pragma must appear after the subprogram declaration, and before any subsequent compilation unit. This pragma specifies the other language (and thereby the calling conventions) and informs the compiler that an object module will be supplied for the corresponding subprogram (see 13.9).
- The only language name allowed is ASSEMBLY. The called procedure or function must conform to the compiler conventions in terms of parameter passage, etc. See pragma IMPORT above and in Appendix F for a related capability.
- LIST** Takes one of the identifiers ON or OFF as the single argument. This pragma is allowed anywhere a pragma is allowed. It specifies that listing of the compilation is to be continued or suspended until a LIST pragma with the opposite argument is given within the same compilation. The pragma itself is always listed if the compiler is producing a listing.
- MEMORY_SIZE** Takes a numeric literal as the single argument. This pragma is only allowed at the start of a compilation, before the first compilation unit (if any) of the compilation. The effect of this pragma is to use the value of the specified numeric literal for the definition of the named number MEMORY_SIZE (see 13.7).
- This pragma has no effect.
- OPTIMIZE** Takes one of the identifiers TIME or SPACE as the single argument. This pragma is only allowed within a declarative part and it applies to the block or body enclosing the declarative part. It specifies whether time or space is the primary optimization criterion.
- This pragma has no effect.
- PACK** Takes the simple name of a record or array type as the single argument. The allowed positions for this pragma, and the restrictions on the named

type, are governed by the same rules as for a representation clause. The pragma specifies that storage minimization should be the main criterion when selecting the representation of the given type (see 13.1).

When Pragma PACK is applied to an array type, and the components are of a discrete type of 8 bits or smaller (e.g., a subtype of INTEGER or an enumeration type), the compiler will pack multiple elements per 16-bit word. The component size in bits is rounded up to a power of 2, i.e., 1, 2, 4, or 8.

When applied to a record type, the compiler will minimize the gaps between discrete components when possible. The compiler will NOT re-order the components to achieve denser packing, and will NOT allocate a component to cross a word boundary (except multi-word components, which always begin and end on a word boundary). If more explicit control over the placement of components is required, a record component representation clause can be used instead.

PAGE

This pragma has no argument, and is allowed anywhere a pragma is allowed. It specifies that the program text which follows the pragma should start on a new page (if the compiler is currently producing a listing).

PRIORITY

Takes a static expression of the predefined integer subtype PRIORITY as the single argument. This pragma is only allowed within the specification of a task unit or immediately within the outermost declarative part of a main program. It specifies the priority of the task (or tasks of the task type) or the priority of the main program (see 9.8).

SHARED

Takes the simple name of a variable as the single argument. This pragma is allowed only for a variable declared by an object declaration and whose type is a scalar or access type; the variable declaration and the pragma must both occur (in this order) immediately within the same declarative part or package specification. This pragma specifies that every read or update of the variable is a synchronization point for that variable. An implementation must restrict the objects for which this pragma is allowed to objects for which each of direct reading and direct updating is implemented as an indivisible operation (see 9.11).

STACK_SIZE Takes a task type name and a static expression of some integer type. This pragma is allowed anywhere that a task storage specification is allowed. The effect of this pragma is to use the value of the expression as the number of storage units (words) to be allocated to the process stack of tasks of the associated task type. (See the 'STORAGE_SIZE length representation clause mentioned in Appendix F for a complementary capability.)

STORAGE_UNIT Takes a numeric literal as the single argument. This pragma is only allowed at the start of a compilation, before the first compilation unit (if any) of the compilation. The effect of this pragma is to use the value of the specified numeric literal for the definition of the named number STORAGE_UNIT (see 13.7).

This pragma has no effect.

SUPPRESS Takes as arguments the identifier of a check and, optionally, also the name of an object, a type or subtype, a subprogram, a task unit, or a generic unit. This pragma is only allowed either immediately within a declarative part or immediately within a package specification. In the latter case, the only allowed form is with a name that denotes an entity (or several overloaded subprograms) declared immediately within the package specification. The permission to omit the given check extends from the place of the pragma to the end of the declarative region associated with the innermost enclosing block statement or program unit. For a pragma given in a package specification, the permission extends to the end of the scope of the named entity.

If the pragma includes a name, the permission to omit the given check is further restricted: it is given only for operations on the named object or on all objects of the base type of a named type or subtype; for calls of a named subprogram; for activations of tasks of the named task type; or for instantiations of the given generic unit (see 11.7).

The implementation only supports the following form:

```
pragma SUPPRESS (check-name);
```

PREDEFINED LANGUAGE PRAGMAS

Page B-6

i.e., it is not possible to restrict the omission of a certain check to a specified type or object.

One of the following check-names can be specified: ACCESS_CHECK, DISCRIMINANT_CHECK, INDEX_CHECK, LENGTH_CHECK, RANGE_CHECK, DIVISION_CHECK, OVERFLOW_CHECK, ELABORATION_CHECK, or STORAGE_CHECK.

SYSTEM_NAME Takes an enumeration literal as the single argument. This pragma is only allowed at the start of a compilation, before the first compilation unit (if any) of the compilation. The effect of this pragma is to use the enumeration literal with the specified identifier for the definition of the constant SYSTEM_NAME. This pragma is only allowed if the specified identifier corresponds to one of the literals of the type NAME declared in the package SYSTEM (see F.3).

This pragma has no effect.

UNIVERSAL_DATA This pragma has no argument and is only allowed immediately within the declarative part of a library package specification or body.

This pragma allows the static data of a library package specification or body to be located anywhere in memory. Otherwise, all static data must be located in a single 64K data segment, together with all user task process stacks. Specifications and bodies of the same unit are treated independently, and the pragma applies only to the compilation unit it appears in. See Appendix F, or Reference 6, for more detail.

APPENDIX C

PREDEFINED LANGUAGE ENVIRONMENT

This annex outlines the specification of the package STANDARD containing all predefined identifiers in the language. The corresponding package body is implementation-defined and is not shown.

The operators that are predefined for the types declared in the package STANDARD are given in comments since they are implicitly declared. Italics are used for pseudo-names of anonymous types (such as *universal real*) and for undefined information (such as *implementation_defined* and *any_fixed_point_type*).

Only the implementation-specific portions of STANDARD are shown below:

package STANDARD is

...

type INTEGER is range -32_768 .. 32_767;

type LONG_INTEGER is range -2_147_483_648 .. 2_147_483_647;

-- The ranges for FLOAT and LONG_FLOAT are approximate here.

type FLOAT is digits 6 range -1.70141E38 .. 1.70141E38;

type LONG_FLOAT is digits 9 range -1.70141183E38 ..
1.70141183E38;

type DURATION IS DELTA 0.0001 range -131_072.0 .. 131_071.0;

...

end STANDARD;

PREDEFINED LANGUAGE ENVIRONMENT

Page C-2

The language definition also defines the following library units:

- The package CALENDAR (see 9.6)
- | - The package SYSTEM (see F.3)
- | - The package MACHINE_CODE (see F.9.3)
- The generic procedure UNCHECKED DEALLOCATION (see 13.10.1)
- The generic function UNCHECKED_CONVERSION (see 13.10.2)
- The generic package SEQUENTIAL_IO (see 14.2.3)
- The generic package DIRECT_IO (see 14.2.5)
- The package TEXT_IO (see 14.3.10)
- The package IO EXCEPTIONS (see 14.5)
- | The package LOW_LEVEL_IO is not provided.

APPENDIX D

GLOSSARY

The following terms are an addition to those found in the Ada Reference Manual (ANSI/MIL-STD-1815A, 22-Jan-83).

- AAMP. Advanced Architecture Microprocessor. One of the CAPS family of stack architecture processors, also known as CAPS-9. It is a high-performance, general-purpose, 16-bit processor with floating point arithmetic on a single chip. It is available in two versions, indicated as AAMP1 and AAMP2 in this document.
- CAPS. Collins Adaptive Processing System. The CAPS series of microprocessors uses a stack architecture that supports the use of high order programming languages for embedded computer systems.

APPENDIX E
SYNTAX SUMMARY

See Ada Reference Manual (ANSI/MIL-STD-1815A, 22-Jan-83).

APPENDIX F

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

This appendix describes the implementation-dependent characteristics of the DDC-Based Ada/CAPS Compiler.

F.1 Implementation-Dependent Pragas.

F.1.1 Pragma EXPORT.

Pragma EXPORT takes an identifier denoting a subprogram or an object, and optionally takes a string literal (the name of a CAPS object allowed at the place of a declarative item and must apply to a subprogram or object declared by an earlier declarative item in the same declarative part or package specification. The pragma must occur in the same compilation unit as the subprogram body to export a subprogram, and in the same compilation unit as the declaration to export an object. The subprogram to be exported may not be nested within anything but a library_unit package specification or body. The pragma is not allowed for an access or a task object. The object exported must be a static object. Generally, objects declared in a package specification or body are static; objects declared local to a subprogram are not.

This pragma makes a subprogram or object visible to non-Ada parts of the system.

```
pragma EXPORT(internal_name [, external_name]);

internal_name ::= identifier

external_name ::= string_literal
```

If external name is not specified, the internal_name is used as the external_name. If a string literal is given, it is used. External name must be an identifier that is acceptable to the CAPS linker, although it does not have to be a valid Ada identifier.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

Page F-2

Exporting Subprograms:

In this case, pragma EXPORT specifies that the body of the specified subprogram associated with an Ada subprogram specification may be called from another CAPS language (e.g., JOVIAL or assembly).

Exported subprograms must be uniquely identified by their internal names. An exported subprogram must be a library unit or be declared in the outermost declarative part of a library package (specification or body). Pragma EXPORT is allowed for a subprogram which is a compilation unit only after the subprogram body in the compilation unit. It is allowed for a subprogram in a package body after the body of the subprogram. Pragma EXPORT is not allowed in a package specification.

Example:

```

procedure BUILT_IN_TEST (MODE: in INTEGER) is
...
end BUILT_IN_TEST;
pragma EXPORT (BUILT_IN_TEST);

```

Exporting Objects:

In this case, Pragma EXPORT specifies that an Ada object is to be accessible by an external routine in another CAPS language.

Exported objects must be uniquely identified by their internal names. An exported object must be a variable declared in the outermost declarative part of a library package (specification or body).

The object must be allocated to static storage. To guarantee this, the subtype indication for the object must denote one of the following:

- o A scalar type or subtype.
- o An array subtype with static index constraints whose component size is static.
- o A simple record type or subtype.

Example:

```

SYSTEM_STATUS : INTEGER;
pragma_EXPORT (SYSTEM_STATUS, "SYSSSTS");
-- SYSSSTS is the external name corresponding to the
-- JOVIAL/AAMP identifier SYS'STS.

```

F.1.2 Pragma IMPORT.

Pragma IMPORT takes an internal name denoting a subprogram, and, optionally, takes an external name (the name of a CAPS object module entry/external name) as arguments. This pragma is only allowed at the place of a declarative item and must apply to a subprogram declared by an earlier declaration item in the same declarative part or package specification.

This pragma allows the import of a procedure or function from a non-Ada environment.

```
pragma IMPORT (internal_name [, external_name]);  
  
internal_name ::= identifier | string_literal  
  
external_name ::= identifier | string_literal
```

Internal_name may only be a string_literal when designating an operator function for import. If external_name is not specified, the internal_name is used as the external_name. If an identifier or string_literal is given, it is used. External_name must name an identifier that is acceptable to the CAPS target linker, though it does not have to be a valid Ada identifier.

Importing Subprograms:

In this case, pragma IMPORT specifies that the body of the specified subprogram associated with an Ada subprogram specification is to be provided by another CAPS language. Pragma INTERFACE must also be given for the internal_name earlier for the same declarative part or package specification. The use of pragma INTERFACE implies that a corresponding subprogram body is not given.

Imported subprograms must be uniquely identified by their internal names. An imported subprogram must be a library unit or be declared in the outermost declarative part of a library package (specification or body). Pragma IMPORT is allowed only if either the body does not have a corresponding specification, or the specification and body occur in the same declarative part.

If a subprogram has been declared as a compilation unit, pragma IMPORT is only allowed after the subprogram declaration and before any subsequent compilation unit. This pragma may not be used for a subprogram that is declared by a generic instantiation of a predefined subprogram.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

Page F-4

Example:

```
function SIN (X: in FLOAT) return FLOAT;  
pragma INTERFACE (ASSEMBLY, SIN);  
pragma IMPORT (SIN, "SIN$$");
```

Importing Objects:

In this case, pragma IMPORT specifies that a non-Ada object is to be made available to an Ada environment.

Objects must be uniquely identified by their internal names. Because it is not created by an Ada elaboration, an imported object cannot have an initial value. This has the following implications:

- o It cannot be a constant (explicit initial value).
- o It cannot be an access type (implicit initial value of null).
- o It cannot be a record type that has discriminants (which are always initialized) or components with default initial values.
- o It cannot be an object of a task type.

Example:

```
IO_FLAG: BOOLEAN;  
pragma IMPORT (IO_FLAG, "IO.1");
```

F.1.3 Pragma STACK_SIZE.

Pragma STACK_SIZE has two arguments: a task type name and an integer expression. This pragma is allowed anywhere that a task storage specification is allowed. The effect of this pragma is to use the value of the expression as the number of storage units (words) to be allocated to the process stack of tasks of the associated task type.

Example:

```

task type DISPLAY_UNIT is
    entry UPPER_DISPLAY;
    entry BOTTOM_LINE;

end DISPLAY_UNIT;

for DISPLAY_UNIT'SORAGE SIZE use 20 000;    -- Data Stack.
pragma STACK_SIZE (DISPLAY_UNIT, 1000);    -- Process Stack.

```

F.1.4 Pragma UNIVERSAL_DATA

Pragma UNIVERSAL_DATA allows the static data of a library package specification or body to be located anywhere in memory. Otherwise, all static data must be located in a single 64K data segment, together with all user task process stacks.

Use of this pragma causes less efficient code (in both space and speed) to be generated by the compiler for most references to data declared in the package to which it applies. Specifications and bodies of the same unit are treated independently, and the pragma applies only to the compilation unit in which it appears.

Pragma UNIVERSAL_DATA must appear in the declarative part of a package specification or body, and must appear before any other object declaration.

The functionality of this pragma is also available as the compiler command option /PRAGMA = UNIVERSAL_DATA.

Example:

```

package DATABASE is
    pragma UNIVERSAL_DATA;

    -- object declarations

    ...

end DATABASE;

```

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

Page F-6

F.2 Implementation-Dependent Attributes.

No implementation-dependent attributes are supported.

F.3 Specification Of The Package SYSTEM.

package SYSTEM is

```

type ADDRESS      is range 0..16#FF FFFF#    -- 24 bit address
subtype PRIORITY  is INTEGER range 1 .. 254;
type NAME         is (AAMP1, AAMP2);
SYSTEM_NAME:      constant NAME      := AAMPx;
STORAGE_UNIT:     constant           := 16;
MEMORY_SIZE:      constant           := 16_384 * 1024;
MIN_INT:          constant           := -2_147_483_647-1;
MAX_INT:          constant           := 2_147_483_647;
MAX_DIGITS:       constant           := 9;
MAX_MANTISSA:     constant           := 31;
FINE_DELTA:       constant           := 2#1.0#E-31;
TICK:             constant           := 0.000_1;

type INTERFACE_LANGUAGE is (ASSEMBLY);

EXCEPTO : exception; -- Raised by EXCEPTO instruction
                    -- (see run-time package MACHINE_EXCEPTIONS).

```

end SYSTEM;

F.4 Representation Clauses.

The representation clauses that are accepted are described below. Note that representation clauses can now be given on derived types, and that the "Change of Representation" feature described in RM 13.6 is supported.

F.4.1 Length Clauses.

Four kinds of length clauses are accepted: size specification, collection size specification, task STORAGE_SIZE specification, and SMALL specification.

F.4.1.1 Size Specification.

The size clause for a type T is accepted in the following cases.

- If T is a discrete type (integer or enumeration), then the specified size must be greater than or equal to the number of bits required to represent the type, and no larger than 32 bits.

Note that when the number of bits needed to hold any value of the type is calculated, the range is extended to include zero if necessary. For example, the range 6 .. 7 cannot be represented in one bit, but requires three bits. A range with a negative lower bound is allowed.

- If T is a fixed-point type, then the specified size must be greater than or equal to the smallest number of bits needed to hold any value of the type, and no larger than 32 bits. Note that the Reference Manual permits a representation where the lower and upper bounds are not included. Thus, the type:

type FIX is delta 1.0 range -1.0 .. 7.0;

is representable in 3 bits. As for discrete types, the number of bits required for a fixed-point type is calculated using the range extended to include 0.0.

- If T is a floating-point type, an access type, or a task type, the specified size must be equal to the number of bits used to represent a value of the type. A floating-point type requires 32 or 48 bits. An access or task type requires 32 bits.
- If T is a record type, then the specified size must be greater than or equal to the number of bits per storage unit (16) times the number of storage units required for an object of the type. In other words, a record cannot occupy a partial storage unit.
- If T is an array type, a size clause is accepted only if the size of the array is static, i.e., known at compile time. The specified size must be greater than or equal to the number of bits required to represent a value of the type.

F.4.1.2 Collection Size Specification.

Using the `STORAGE_SIZE` clause on an access type will cause the specified number of storage units to be allocated from the heap and set aside for future allocations for the access type. The maximum size of a collection is limited only by the size of the heap, which, in turn, is limited by the machine's 24-bit address space.

F.4.1.3 Task `STORAGE_SIZE` Specification.

When the `STORAGE_SIZE` clause is given for a task type, the data stack for each object of the task type will be the specified size. The default storage size for a task, and for the main program and the primal task, are specified in the linker command (`.LEC`) file.

F.4.1.4 `SMALL` Specification.

Any value equal to or smaller than the specified delta can be given as the `SMALL` specification for a fixed point type, as long as the value is either an integer (e.g. 1.0, 1000.0), or the reciprocal of an integer (e.g. 0.5, 0.2, 0.01, 1.0/180.0, $2.0^{*(-15)}$). Other fractions are not allowed, e.g. 2.0/3.0, 0.375.

F.4.2 Enumeration Representation Clauses.

An enumeration representation clause may specify representation values in the range `INTEGER'first .. INTEGER'last`. An enumeration representation clause may be combined with a length clause. If a representation clause has been given for an enumeration type, the representational values are considered when the number of bits needed to hold any value of the type is computed. Thus, the type:

```
type COLOR is (RED, WHITE, BLUE);
for COLOR use (1, 4, 7);
```

requires three bits when the representation clause is given, but only two bits when it is omitted.

F.4.3 Record Representation Clauses.

When component clauses are applied to a record type, the following restrictions are imposed.

- All values of the component type must be representable within the specified number of bits in the component clause.
- If the component type is a discrete type or a fixed-point type, then the component is packed into the specified number of bits. The component is allowed to span a word boundary (although operations will be less efficient), but a component larger than 16 bits is not allowed to cross TWO word boundaries.
- If the component type is a PACKED array type with a discrete or fixed-point element type, then the component is packed into the specified number of bits. However, the first component must begin at an element-size boundary, to avoid the possibility of any component crossing a word boundary.

Pragma PACK must be specified for an array type if a record component of that type has a component clause where the given size in bits requires a packed representation.

- If the component type is not one of the types mentioned above, it must begin at a word boundary, and the default size for the type must be specified as the bit width.

Two components are not allowed to overlap (unless they are in different variants in a discriminant record), and the compiler is required to report an error if such a representation is given.

If the record type contains components which are not mentioned in a component clause, they are allocated consecutively beginning at the next storage unit following the component with the highest specified bit offset. If pragma PACK is applied to the record type, these components will be packed as described under Pragma PACK in Appendix B, otherwise each will begin at a storage unit boundary.

If there are gaps between components, due either to a representation clause or pragma PACK, all objects of the record type will be block-filled with zero by the compiler to allow efficient record comparison.

F.4.4 Alignment Clauses.

Only a clause specifying record alignment at a storage unit is accepted (for REC_TYPE use record at mod 1;), as no larger alignment boundaries are meaningful in the AAMP architecture.

F.4.5 Implementation-Generated Names.

Implementation-generated names for implementation-dependent components are not supported, as no "invisible" components of a record type are ever generated by the compiler.

F.5 Address Clause Expressions.

Address clauses for objects are accepted. The type ADDRESS in package SYSTEM is simply an integer type; therefore, the following kinds of address clauses are allowed:

```
for X use at 16#12_3456#;  
for Y use at X'address + 3;
```

The address of a data object is interpreted as the 24-bit address of a storage unit (16-bit word).

Address clauses for subprograms, tasks, and packages are not supported.

F.6 UNCHECKED_CONVERSION Restrictions.

UNCHECKED_CONVERSION is only allowed between objects of the same size.

F.7 I/O Package Implementation-Dependent Characteristics.

The target environment does not support a file system; therefore, I/O procedure or function calls involving files (except STANDARD_OUTPUT, etc. as noted below) will raise an exception.

F.7.1 Package SEQUENTIAL_IO.

All procedures and functions raise STATUS_ERROR, except for CREATE and OPEN which raise USE_ERROR, and IS_OPEN which always returns FALSE.

F.7.2 Package DIRECT_IO.

All procedures and functions raise STATUS_ERROR, except for CREATE and OPEN which raise USE_ERROR, and IS_OPEN which always returns FALSE.

F.7.3 Package TEXT_IO.

No disk file system is supported. Therefore, procedures CREATE and OPEN always raise USE_ERROR.

The output routines with no file parameter, which operate on the current default output file, are implemented and produce their output via package LOW_IO. Since no external files can be opened, the output routines with a file parameter raise STATUS_ERROR unless the actual parameter is one of the functions STANDARD_OUTPUT or CURRENT_OUTPUT.

Similarly, the input routines with a file parameter raise STATUS_ERROR unless the parameter is STANDARD_INPUT or CURRENT_INPUT.

Function IS_OPEN returns TRUE if the parameter is one of STANDARD_INPUT, STANDARD_OUTPUT, CURRENT_INPUT, or CURRENT_OUTPUT; otherwise FALSE is returned.

F.7.4 Package LOW_LEVEL_IO.

Package LOW_LEVEL_IO is not provided.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

Page F-12

F.7.5 Package LOW_IO.

Package LOW_IO is used by TEXT_IO for character-level I/O operations. The implementation provided with the compiler system sends output messages to the Symbolic Debugger to be printed on the screen and optionally sent to a .LOG file. The Debugger does not implement input.

The body of LOW_IO can be replaced by the user, for example, to communicate with a terminal via an RS-232 port.

F.8 Other Implementation-Dependent Features.

F.8.1 Predefined Types.

This section describes the implementation-dependent predefined types declared in the predefined package STANDARD, and the relevant attributes of these types.

F.8.1.1 Integer Types.

Two predefined integer types are implemented: INTEGER and LONG_INTEGER. They have the following attributes:

INTEGER'FIRST	= -32768
INTEGER'LAST	= 32767
INTEGER'SIZE	= 16
LONG_INTEGER'FIRST	= -2_147_483_648
LONG_INTEGER'LAST	= 2_147_483_647
LONG_INTEGER'SIZE	= 32

F.8.1.2 Floating Point Types.

Two predefined floating point types are implemented: FLOAT and LONG_FLOAT. They have the following attributes:

FLOAT'DIGITS	= 6
FLOAT'EMAX	= 84
FLOAT'EPSILON	= 16#0.1000_000#E-04
	~ = 9.53674E-07
FLOAT'FIRST	= -16#0.7FFF_FF8#E+32
	~ = -1.70141E+38
FLOAT'LARGE	= 16#0.FFFF_F80#E+21
	~ = 1.93428E+25

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

Page F-13

FLOAT'LAST	=	16#0.7FFF FF8#E+32
	~	1.70141E+38
FLOAT'MACHINE_EMAX	=	127
FLOAT'MACHINE_EMIN	=	-127
FLOAT'MACHINE_MANTISSA	=	24
FLOAT'MACHINE_OVERFLOWS	=	TRUE
FLOAT'MACHINE_RADIX	=	2
FLOAT'MACHINE_ROUNDS	=	TRUE
FLOAT'MANTISSA	=	21
FLOAT'SAFE_EMAX	=	127
FLOAT'SAFE_LARGE	=	16#0.7FFF FC#E+32
	~	1.70141E+38
FLOAT'SAFE_SMALL	=	16#0.1000 000#E-31
	~	2.93874E-31
FLOAT'SIZE	=	32
FLOAT'SMALL	=	16#0.8000 000#E-21
	~	2.58494E-26
LONG_FLOAT'DIGITS	=	9
LONG_FLOAT'EMAX	=	124
LONG_FLOAT'EPSILON	=	16#0.4000 0000 000#E-7
	~	9.31322575E-10
LONG_FLOAT'FIRST	=	-16#0.7FFF FFFF FF8#E+32
	~	-1.70141183E+38
LONG_FLOAT'LARGE	=	16#0.FFFF FFFE 000#F+31
	~	2.12676479E+37
LONG_FLOAT'LAST	=	16#0.7FFF FFFF FF8#E+32
	~	1.70141183E+38
LONG_FLOAT'MACHINE_EMAX	=	127
LONG_FLOAT'MACHINE_EMIN	=	-127
LONG_FLOAT'MACHINE_MANTISSA	=	40
LONG_FLOAT'MACHINE_OVERFLOWS	=	TRUE
LONG_FLOAT'MACHINE_RADIX	=	2
LONG_FLOAT'MACHINE_ROUNDS	=	TRUE
LONG_FLOAT'MANTISSA	=	31
LONG_FLOAT'SAFE_EMAX	=	127
LONG_FLOAT'SAFE_LARGE	=	16#0.7FFF FFFF#E+32
	~	1.70141183E+38
LONG_FLOAT'SAFE_SMALL	=	16#0.1000 0000 000#E-31
	~	2.93873588E-39
LONG_FLOAT'SIZE	=	48
LONG_FLOAT'SMALL	=	16#0.8000 0000 000#E-31
	~	2.35098870E-38

F.8.1.3 Fixed Point Types.

To implement fixed point numbers, Ada/CAPS uses two sets of anonymous, predefined, fixed point types, here named `FIXED` and `LONG FIXED`. These names are not actually defined in package `STANDARD`, but are used here only for reference.

These types are of the following form:

```
type FIXED_TYPE is delta SMALL range -M*SMALL .. (M-1)*SMALL;
```

where $SMALL = 2^n$ for $-128 \leq n \leq 127$,
and $M = 2^{15}$ for `FIXED`, or $M = 2^{31}$ for `LONG_FIXED`.

For each of `FIXED` and `LONG_FIXED`, there exists a virtual predefined type for each possible value of `SMALL` (see RM 3.5.9). `SMALL` may be any power of 2 which is representable by a `LONG FLOAT` value. `FIXED` types are represented by 16 bits, while 32 bits are used for `LONG_FIXED` types.

A user-defined fixed point type is represented as that predefined `FIXED` or `LONG_FIXED` type which has the largest value of `SMALL` not greater than the user-specified `DELTA`, and which has the smallest range that includes the user-specified range.

As the value of `SMALL` increases, the range increases. In other words, the greater the allowable error (the value of `SMALL`), the larger the allowable range.

Example 1:

For a 16-bit `FIXED` type, to get the smallest amount of error possible requires $SMALL = 2^{(-128)}$, but the range is constrained to:

$$-(2^{15}) * 2^{(-128)} .. (2^{15} - 1) * 2^{(-128)}, \text{ which is } \\ -(2^{(-113)}) .. 2^{(-113)} - 2^{(-128)}.$$

Example 2:

For a `FIXED` type, to get the largest range possible requires $SMALL = 2^{127}$, i.e., the error may be as large as 2^{127} . The range is then:

$$-(2^{15}) * (2^{127}) .. (2^{15} - 1) * (2^{127}), \text{ which is } \\ -(2^{142}) .. (2^{142}) - (2^{127}).$$

Example 3:

Two particularly useful fixed-point types range from -1.0 to 1.0 - delta. The compiler is able to take advantage of special AAMP instructions to generate more efficient code for multiplication and division than for fixed-point types in general.

```
type FRACT      is delta 2.0**(-15) range -1.0 .. 1.0 - 2.0**(-15);
```

```
type LONG_FRACT is delta 2.0**(-31) range -1.0 .. 1.0 - 2.0**(-31);
```

The package FRACTIONAL ARITHMETIC, available in the directory ADACS:[APPLICATION_LIBRARY], defines these types along with useful constants and operations. NOTE_06 in ADACS:[DOCUMENTS] describes how to use the package.

For any FIXED or LONGFIXED type T:

```
T'MACHINE_OVERFLOWS = TRUE
T'MACHINE_ROUNDS    = FALSE
```

F.8.1.4 The Type DURATION.

The predefined fixed point type DURATION has the following attributes:

```
DURATION'AFT          = 4
DURATION'DELTA        = 0.0001
DURATION'FIRST        = -131_072.0000
DURATION'FORE         = 7
DURATION'LARGE        = 131_071.999938965
                     = 2#1.0#E+17 - 2#1.0E-14
DURATION'LAST         = DURATION'LARGE
DURATION'MANTISSA     = 31
DURATION'SAFE_LARGE   = DURATION'LARGE
DURATION'SAFE_SMALL   = DURATION'SMALL
DURATION'SIZE         = 32
DURATION'SMALL        = 6.103515625E-5
                     = 2#1.0#E-14
```

F.8.2 Uninitialized Variables.

There is no check on the use of uninitialized variables. The effect of a program that uses the value of such a variable is undefined.

F.8.3 Package MACHINE_CODE.

Machine code insertions (see RM 13.8) are supported by the Ada/CAPS compiler via the use of the predefined package MACHINE_CODE.

package MACHINE_CODE is

```

    type CODE is record
        INSTR: STRING (1 .. 71);
    end record;
```

end MACHINE_CODE;

Machine code insertions may be used only in a procedure body. No local declarations are allowed, except for USE clauses. The body must contain nothing but code statements, as in the following example:

```

with MACHINE_CODE; -- Must apply to the compilation unit
                    -- containing DOUBLE.
```

```

procedure DOUBLE (VALUE: in INTEGER; DOUBLED_VALUE: out INTEGER) is
    use MACHINE_CODE;
begin
```

```

    CODE' (INSTR => "    REFSL  1  ;"); -- Get VALUE.
    CODE' (INSTR => "    DUP      ;"); -- Make a copy of VALUE.
    CODE' (INSTR => "    ADD      ;"); -- Add copies together.
    CODE' (INSTR => "    ASNSL  0  ;"); -- Store result in
                                     -- DOUBLED_VALUE.
```

end DOUBLE;

The string literal assigned to INSTR may be any CAPS assembly language instruction or macro. The string is written directly to the assembly output file.

The file AAMPx.MAC (where x is 1 or 2), located in the TOOLS subdirectory of the compiler system, defines the instructions and macros which are available for use. The macros may change with different compiler system releases and should be used cautiously, as there is no guarantee that they will perform the same in future releases. The CAPS Macro Assembler User's Guide contains information on how to call macros and write assembly instructions.

Several application notes are available in ADACS:[DOCUMENTS] which explain in greater detail how to use machine-code insertions, including clever ways to implement efficient built-in functions.

F.8.4 Compiler Limitations.

The following limitations apply to Ada programs in the DDC-Based Ada/CAPS Compiler System:

- o A compilation unit may not contain more than 64K bytes (32K words) of code.
- o A compilation unit may not contain more than 32K words of volatile data.
- o A compilation unit may not contain more than 32K words of constant data.
- o The local environment of a subprogram is limited to 32K words.
- o It follows that any single object may be no larger than 32K words.
- o Arrays are limited to 32K elements, even if packed.
- o No more than 500 subprograms may be defined in a single compilation unit, including any implicitly created by the compiler.
- o The maximum nesting level for blocks is 100.